ESD-TDR-65-36

Technical Report 377

An Experimental On-Line Data Storage and Retrieval System

J. F. Nolan
A. W. Armenti

3 February 1965

ESRL

Prepared under Electronic Systems Division Contract AF 19(628)-500 by

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Lexington, Massachusetts

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology, with the support of the U.S. Air Force under Contract AF 19(628)-500. The computer time was supported by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01).

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LINCOLN LABORATORY

AN EXPERIMENTAL ON-LINE DATA STORAGE AND RETRIEVAL SYSTEM

J. F. NOLAN

A. W. ARMENTI

Group 28

TECHNICAL REPORT 377

3 FEBRUARY 1965

LEXINGTON MASSACHUSETTS

ABSTRACT

This report describes an experimental program system designed to test and demonstrate on-line storage and retrieval of formatted data based on complete internal descriptions of the files. The use of internal descriptions allows each user (who need not be a trained programmer) to define, modify, and cross-associate data files to suit his particular needs. The experimental program system was implemented by remote use of the Compatible Time-Sharing System (CTSS) facilities of Project MAC at the Massachusetts Institute of Technology.

Accepted for the Air Force
Stanley J. Wisniewski
Lt Colonel, USAF
Chief, Lincoln Laboratory Office

TABLE OF CONTENTS

AN EXPERIMENTAL ON-LINE DATA STORAGE AND RETRIEVAL SYSTEM

I. INTRODUCTION

This report describes an experimental program system designed to test and demonstrate on-line storage and retrieval of formatted data based on complete internal descriptions of the files. The use of internal descriptions allows each user (who need not be a trained programmer) to define, modify, and cross-associate data files to suit his particular needs. The experimental program system was implemented by remote use of the Compatible Time-Sharing System (CTSS) at M. I. T.

In recent design of computer systems, increasing attention has been paid to the need of users to have easier access to the computer. It has been recognized that processing each user's job to completion without the user's interaction results in inefficient employment of the user's time, wasted processing, long turn-around time, and excessive delay to final problem solution. Considerations such as these have led to the development of facilities in which multiple users have time-shared access to a general processing system.[1] The Project MAC CTSS at M. I. T. is one such system.[2] Users of the CTSS system have direct, on-line access by way of input-output consoles (e.g., teletype stations) to a large-scale computer. Since the individual user's response time per step is typically much longer than the required processing time, he appears to have full power of the processor as if it were his own.

The CTSS system has been used principally by experienced programmers and provides them with a rapid, easy way to prepare, compile, debug, and run programs. But it is apparent that multiple-user systems like CTSS can provide data processing service to the nonprogrammer as well as the programmer and, when equipped with specially designed program systems, can give solutions to problems or answers to questions without requiring the user to write a program. Indeed it is clear that future developments in time-sharing will be characterized not only by a sharing of computer capacity but by a time-sharing of information systems as well. Systems are already under consideration which would make the services of a large library of programs and data files easily available to a great number of users. Professor Corbato, one of the principal designers of the M. I. T. time-sharing system, has described a future system as a multi-user, multi-processor, multi-channel system with multiple access to a vast common structure of data and program procedures for every user.[1]

A critical component of the program system for such a multi-user facility is a set of general-purpose data storage and retrieval programs. Such programs would respond to a broad class of user demands and would be capable of searching voluminous stores of data, technical abstracts, library routines, etc. The ideal system should permit a user to ask factual questions

written in a relatively unrestricted, natural language. The file structure should admit a high degree of variability in format and cross referencing between files. The system should require the user to have little or no knowledge of the file structure. The retrieval logic should give rapid response to storage and retrieval requests, permit a high degree of complexity in the conditions of of search, and require little or no program change as a consequence of file modification. In addition, it is clearly important that both the file structure and retrieval logic allow for an open-ended storage capacity; the system should be able to call upon files from many levels of auxiliary storage (automatically organized according to volume and frequency of access) wherever the search requires it.

The extreme test of the flexibility of such a data storage and retrieval system is in the handling of formatted files where the individual data fields differ in length, coding, and interpretation. The handling of textual data files is a special (and simpler) case of formatted files in which symbol sequences are considered as variable-length fields of consistent coding throughout.

In any system designed to address the above goals, a central problem is the providing of internal file descriptions to accompany the stored data. The system must maintain information on the logical composition of files into entries, the corresponding organization of individual data fields within entries, the classification of data fields as to type, length, coding, units, etc., and, for maximum user utility, the cross-associations of data within and across files. Only if this information is available to the data storage and retrieval programs can the user easily review, define, or query files, and cross-associate data in unanticipated combinations without understanding the internal structure of the filing system. Careful design of the method of internal file description within the system can allow an open-ended variability in file types and formats without requiring modification to the storage and retrieval programs.

In this report, we describe an experimental set of programs which was constructed to test and demonstrate an on-line data storage and retrieval system based upon complete internal description of formatted files. The system assumes an open-ended library of user's files which can be called from auxiliary storage for processing by a name or number designation for each separate set of files. Each file set contains, in addition to the stored data, the file description information required for its interpretation and processing. Within any set of files the user is free to redefine, modify, augment, delete, and cross-associate data as his interests dictate. The level of versatility built into the system can be seen from the following list of its principal features:

(a) The file set can contain many files, each having its own content and logical organization.

(b) The user never need be concerned about the length of his files. All files are variable in length. The total amount of information to be stored is only limited by the total storage capacity.†

(c) All formatting is done by the system. The user refers to file contents by name only. The user is therefore relieved of any need to know about the specific structure of his files.

(d) The user can redefine or modify files on-line, using ordinary English commands and referring to all files or parts of files by name.

†The amount of data in a single file set in the present system is core limited, but this is not an inherent feature of the system.

(e) The user can establish relations between files or parts of files. This allows him or organize the same set of data under a number of category headings or data from a number of different files under a single category heading. This multiple association of data is automatically carried out by the use of list structures; no duplicate storage of data is necessary.

(f) The system prints instructions at the request of the user wherever instructions might be needed. The user does not have to refer to a manual or code list. A new user can begin to use the system without any prior training or any knowledge of programming.

(g) The system detects most frequent clerical errors such as misspellings or incorrect file identification and permits the user to make corrections immediately.

(h) The system provides the user with descriptions about any of the files or relations in the system. This can be a great convenience to the user who has created a number of files and/or data associations and does not remember which relate to the information he wants.

A file in this system is a set of entries. Each entry consists of a set of data fields which describe one of the objects covered by the file name. For example, consider a Personnel file. The entries in such a file describe individuals and each entry will include data fields like the individual's name, his age, his weight, date of birth, etc. In this system, any file may have an arbitrary number of data fields/entry; however, for any given file, every entry must have the same data fields. If the user, for example, wants to include "occupation" as one of the data fields in his Personnel file, then the "occupation" data field must appear in every entry of the file. We may refer to such a file as a fixed-format file. The number of data fields per entry, the number of entries per file, and the number of files are all variable.

At first appearance, the fixed-format rule (requiring the same set of data fields per entry for every entry in a given file) may appear as a limitation on the format of files which can be handled by the system. Most data filing problems do require additional flexibility in three important forms: the variable-length data field, the repeated field, and the subfile. All three deviations from a fixed-format file are allowed in the system.

The variable-length field may be illustrated by the common example of a "remarks" field in filed entries, where an arbitrary-length text sequence may occur. All such variable-length fields are automatically replaced in this system by a fixed-length pointer to an address in a common pool of variable-length data.

The repeated field may be illustrated by the example of multiple telephone numbers per individual in a telephone directory. Here the field is of fixed length but the number of different values to be stored is variable. This is actually a special case of a subfile in which the subfile consists only of a simple list. In general, a subfile consists of an arbitrary number of entries of different content than the entries to which they belong. For complete generality, it is necessary and sufficient to allow subfiles themselves to have subfiles to any number of levels. This capability is provided in the system by the use of relations, which allow the user to associate any two files, not only for the simple file-subfile relationship but also for more complex cross-association of data.

The user creates an association between any two files by defining the relation and giving it a name. A relation associates with each entry of one file a set of entries from a second file. The first file is called the "parent" file and the second file is called the "linkee" file.

An entry from the parent file is called the "parent" entry and the subset of entries from the linkee file with which it is associated is called its "subfile." The relation between a subfile and a

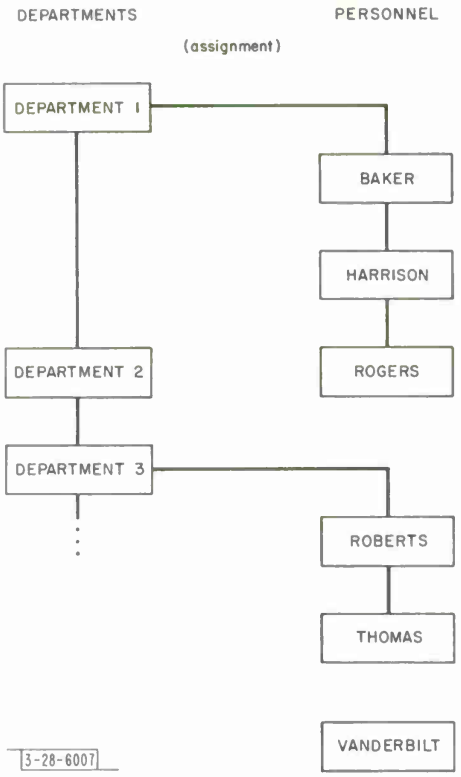


Fig. 1. Example of subfile relation.

parent file requires the subsets of linkee entries to be disjoint (i.e., each entry of the linkee file has at most one parent entry).†

Figure 1 illustrates a subfile relation.‡ Two files are shown, one containing entries describing departments in an organization, the other containing entries describing personnel who are assigned to the departments. These two files are associated by a relation called "assignment." Under this relation, the individuals named Baker, Harrison, and Rogers are associated with Department 1, Roberts and Thomas are associated with Department 3, etc. A parent entry need have no entries in the linkee file associated with it. Conversely, a linkee entry need not be related to any parent entry. Thus, Department 2 has none of this class of personnel assigned to it and Vanderbilt is unassigned.

When a relation is defined between two files, the system associates the subfile entries according to some ordering rule on one of the data fields of the subfile. The user is asked by the system to give the ordering rule and the field on which the ordering is to be done. Under the assignment relation of Fig. 1, for example, the personnel could be ordered alphabetically by last name for each department (as illustrated) or numerically by age (if age is one of the data fields in the Personnel file).

---

† A method for relating nondisjoint subsets of linkees to parents has been worked out but not implemented in the model. We have called this type of relation a "group" relation. The technique for implementing it is described in Sec. V.

‡ We have deliberately kept the files very simple for our example.

The ability of a user to relate files in the system and to search on these relations is one of the powerful features of the system. It gives the user great latitude in establishing cross references between files and getting responses to queries from different files. The user is free to create as many files and as many relations as he chooses. He can therefore cross reference the same files in many different ways if this serves his purpose. Each relation is independent of the others. A file may take part in any number of different relations either as parent or linkee. The same file can be both parent and linkee in a relation. Multiple users can define relations on one another's files without interfering with each other. The one restriction here is that each file name and each relation name be unique to the file set. Similarly, within a file, data field names must be unique. The system responds to a violation of these rules by printing an appropriate error message and requesting the user to provide an alternative name.

The experimental model was implemented on the Project MAC CTSS and is now operational. Since it is an experimental system, it operates as an independent user program; it is not a part of the CTSS program facilities on call to all users. Since we were among 100 users, our quota of disk CTSS storage was limited; therefore, we deliberately adopted a few restrictive conventions. These conventions will be made clear later in the report.

In Sec. II, we detail the system operations from the user's point of view. In Sec. III, we describe the internal makeup of the files and procedures employed in operating on them. Section IV describes the efficiency of the system in the use of storage and in processing time. Finally, in the Sec. V, we indicate some limitations of the experimental model and how it could be extended into a fully developed system.

II. SYSTEM AS SEEN BY USER

The user has control of the data storage and retreival system through a set of commands which he can type at a console. The current repertoire of commands falls into five broad classes.

(a) Control Commands
(b) Commands for Manipulating Files
(c) Commands for Manipulating Data Relations
(d) Commands Block Data Transfer Fields
(e) Block Commands

A. Control Commands

Four commands currently exist which provide the user with general control over the system operations. To find out all the commands that are available to him, the user types *CHOICES. The command *DONE is used in various places to indicate the termination of a list of inputs.

The command *RETURN is used to allow the user to exit from a particular sequence and return to command status.

At the start of operation, the user specifies the tape number on which the file set of interest is stored. Different sets of files may be used, each with its own tape number. The command *START AGAIN permits the user to have a new set of files read into core storage.

B. Commands For Manipulating Files

These commands permit the user to define a new file, enter data into existing file, search a file, delete files, and obtain a description of any file in the current set of active files.

```
*DEFINE FILE                                                    -28-6008
FILE DEFINITION. DO YOU WANT INSTRUCTIONS:YES
PROVIDE THE FOLLOWING(12 CHARACTER MAXIMUM FOR EACH):
NAME OF FILE
TYPE OF FILE(NAMED OR NUMBERED)
THE NAME OF EACH DATA FIELD FOLLOWED BY ITS CODING
ACCEPTABLE CODINGS ARE THE FOLLOWING:
BCD,INTEGER,FLT. POINT,BCD LIST,INTEGER LIST,FLT. PT. LST

A CARRIAGE RETURN MUST FOLLOW EACH INPUT TERM.
THE WORD *DONE TERMINATES INPUT.

COMPUTER
NAMED
RENTAL
INTEGER
ADD TIME
FLT. POINT
CYCLE TIME
FLT. POINT
CORE STORAGE
INTEGER
DRUM STORE
INTEGER
WORD SIZE
BCD
SPEC FEATURE
BCD LIST
*DONE


THE INPUT TABLE FOLLOWS:

COMPUTER       NAMED
RENTAL         INTEGER
ADD TIME       FLT. POINT
CYCLE TIME     FLT. POINT
CORE STORAGE   INTEGER
DRUM STORE     INTEGER
WORD SIZE      BCD
SPEC FEATURE   BCD LIST


IS THIS WHAT YOU WANT. IF NOT,TYPE 'NO' AND START AGAIN.
YES
FILE SET-UP COMPLETED.


COMMAND EXECUTED.


GIVE COMMAND OR TYPE *CHOICES
```

Fig. 2. Example of response to *DEFINE FILE.

To define a file, the user types the command *DEFINE FILE.† The system then asks the user to provide a file name, the name of each data field, the coding type for each data field, and the file type. The six coding types now handled by the system are BCD, integer and floating point and, for repeated fields, lists of these three types. The file type, which is named or numbered, distinguishes files in which the entry name is alphabetic or numeric.

Figure 2 is an example of a dialogue between a user and the system following the command *DEFINE FILE.‡ The file being defined is one which will contain the name and major characteristics of commercial computers and is called "Computer."

The operations of defining a file and entering data into a file are independent. A file must already have been defined before data can be entered into it. However, a user can define a file without entering data. In this case, the file is empty and all that appears in the system is the file description (the file name, its data fields, field coding, etc.).

To enter data into a defined file, the user types the command *INPUT ENTRIES. (This command is useful for entering a small file or modifying files on-line; later commands will describe the method of entering large files from tape recordings under on-line user control.) The system asks the user to provide the file name. It then types the name of each data field and the coding specification for each.. The user then types the data values for each data field of an entry. When the last data value has been typed, the completed entry is entered into the file. As many entries as desired can be added to the file in this way.

If relations have been defined on the file, the name of each is typed by the system and the user must then give the name of each parent entry (if one exists) for the data entry being added to the file.

Figure 3 is an example of the system response to the command *INPUT ENTRIES. In the example, entries are stored in the Computer file set up by the *DEFINE FILE command illustrated in Fig. 2.

To search a file, the user types the command *SEARCH FILE. The system types out a list of the files that are currently active and asks the user to name the file to be searched. When this is given, the system types a brief description of the file and provides the user with a sample entry. The user is then asked to provide the data field on which the search is to be made, the conditions of the search, and the reference or test value for the condition. At present, the system will search on the conditions "equal to," "less than," "greater than," "less than or equal to," and "greater than or equal to."

Figure 4 is an example of the system response to the command *SEARCH FILE. The search was performed on the Computer file. Two searches were conducted: one asked for full entries to be printed on computers having a cycle time less than 4.0 μsec; the other asked for the name only for computers with a core storage greater than 32,000 words.

The brief description of the file printed before the search request is an aid to the user in formulating his search conditions. A separate command *DESCRIBE FILE can also be used to get such a file description. By making a file description available, the system relieves the user of the need to remember the data field names or general content of a file.

In addition to the command for describing a file, defining a file, entering data, and searching a file, the system includes the following:

---

† Commands are distinguished from other inputs by a leading asterisk(*).

‡ In this example and in those that follow, the information typed by the user is underlined.

```
*INPUT ENTRIES                                                  -28-6009

TYPE:
FILE NAME
*INSTRUCTIØNS ØR *NØ

CØMPUTER
*INSTRUCTIØNS


 FØR EACH ENTRY TØ BE ADDED:
       1. WAIT UNTIL 'READY' IS TYPED
       2. LIST CØNTENTS ØF THE DATA FIELDS
             A. IF SØME FIELD IS ITSELF A LIST,
                A BLANK LINE SIGNIFIES THE END ØF THE LIST
             B. FØRMATS ARE:
                   FØR BCD : FIELD LENGTH=6, LEFT JUSTIFY DATA
                   FØR INTEGERS : FIELD LENGTH=12, RIGHT JUSTIFY DATA
                   FØR FLT. PT. : FIELD LENGTH=16, PRØVIDE DECIMAL PT.
       3. TYPE THE PARENT ØF THIS ENTRY FØR EACH RELATIØN LISTED
       4. TØ TERMINATE INPUT ØF ENTRIES, PRESS CR AFTER 'READY' IS TYPED


DATA FIELDS
  NAME              CØDING
NAME                BCD
ADD TIME            FLØATING PØINT
CØRE STØRAGE        INTEGER
CYCLE TIME          FLØATING PØINT
DRUM STØRE          INTEGER
RENTAL              INTEGER
SPEC FEATURE        BCD LIST
WØRD SIZE           BCD

RELATIØNS
THERE ARE NØ RELATIØNS


READY

IBM 7094 II
1.4
          32
1.4
          186
          160
IN'RUP
16XR'S
FLT.PT
IN'ADD

64B
```

Fig. 3. Example of response to *INPUT ENTRIES.

```
READY
CDC 3600
2.0
          262
1.5
0.0
          55
IN'RUP
6XR'S
FLT.PT
IN'ADD

48B


 READY
UNIVAC 1107
4.0
          65
4.0
0.0
          45
IN'RUP
15XR'S
FLT.PT
IN'ADD

36B


 READY
DEC PDP-6
3.5
          262
4.0
          96
           8
IN'RUP
15XR'S
FLT.PT
IN'ADD

36B


READY
DEC PDP-1
10.0
          65
5.0
         131
           3
IN'RUP
IN'ADD

18B
```

Fig. 3. Continued.

```
*SEARCH FILE                                                    -28-6010

    THE ACTIVE FILES ARE :

CØMPUTER
HØME ADDRESS
STREET


PRØVIDE FILE NAME:CØMPUTER


              (FILE DESCRIPTIØN)


CØMPUTER      IS A FILE WITH    NAMED ENTRIES.
NØ. ØF DATA FIELDS PER ENTRY=  7


SAMPLE ENTRY FØLLØWS:

ENTRY: CDC 3600
  ADD TIME     :     2.00
  CØRE STØRAGE:    262
  CYCLE TIME   :     1.50
  DRUM STØRE   :      0
  RENTAL       :     55
  SPEC FEATURE: IN'RUP
                6XR'S
                FLT.PT
                IN'ADD
  WØRD SIZE    : 48B


              (START ØF SEARCH)


PRØVIDE FIELD NAME:      CYCLE TIME
PRØVICE CØNDITIØN(EQ,LT,GT,LTØREQ,GTØREQ):LT
PRØVIDE TEST VALUE(FLTG. PØINT NUMBER):    4.0
DØ YØU WANT FULL ENTRIES PRINTED:    YES


(START ØF SUBFILE)

ENTRY: CDC 3600
  ADD TIME     :     2.00
  CØRE STØRAGE:    262
  CYCLE TIME   :     1.50
  DRUM STØRE   :      0
  RENTAL       :     55
  SPEC FEATURE: IN'RUP
                6XR'S
                FLT.PT
                IN'ADD
  WØRD SIZE    : 48B
```

Fig. 4. Example of response to *SEARCH FILE.

```
ENTRY: IBM 7094 II                                          -28-6010
  ADD TIME    :      1.40
  CØRE STØRAGE:      32
  CYCLE TIME  :      1.00
  DRUM STØRE  :     186
  RENTAL      :     160
  SPEC FEATURE:  IN'RUP
                 16XR'S
                 FLT.PT
                 IN'ADD
  WØRD SIZE   :  64B

(END ØF SUBFILE)
NØ. ØF MATCHING ENTRIES=     2


              (START ØF SEARCH)


PRØVIDE FIELD NAME:      CØRE STØRAGE
PRØVIDE CØNDITIØN(EQ,LT,GT,LTØREQ,GTØREQ):GT
PRØVIDE TEST VALUE(6 DIGITS):      32
DØ YØU WANT FULL ENTRIES PRINTED:   NØ


(START ØF SUBFILE)
CDC 3600
DEC PDP-1
DEC PDP-6
UNIVAC 1107

(END ØF SUBFILE)
NØ. ØF MATCHING ENTRIES=     4


              (START ØF SEARCH)


PRØVIDE FIELD NAME:      *RETURN


CØMMAND EXECUTED.


GIVE CØMMAND ØR TYPE *CHØICES
```

Fig. 4. Continued.

(1) *LIST FILES. The system lists all active files by name.

(2) *PRINT FILE. The system prints out the entry name of each entry in the file specified by the user.

(3) *FIND VALUE. The system gives the data value for a particular data field in a file. The user is asked to specify the name of the file, the name of the entry, and the data field.

(4) *DELETE FILE. The system removes the designated file from the system.

C. Commands For Manipulating Relations

These commands permit the user to establish and name relations between any two active files, to search files under an already defined relation, obtain parent or linkee entries for selected relations, and to delete existing relations. A relation, it will be recalled, associates with each entry of a "parent" file a subset of entries from a "linkee" file.

To define a relation, the user first types the command *DEFINE RELATION. The system then asks the user to provide the name of the parent file, the name of the linkee file, and the ordering field. If both the parent file and the linkee file have more than one entry, the system requires the user to specify for each file which entries are to be associated. To use our earlier example, if individuals in a Personnel file are to be associated with departments in the Department file under a relation called "assignment," then the user must identify the set of individuals to be assigned to each department. If either file is empty, however, the system simply sets up the required programming structure for the relation. The user will be asked to make the proper associations between entries at the time they are entered into the files.† In the case of batched data inputted from tape, the parent names can appear as part of each data record. This latter technique would be the normal method of establishing standard file-subfile relationships when large files are initially entered into the system.

Figure 5 is an example of an exchange between the user and system for the command *DEFINE RELATION. The example shows the former case where file entries already exist and the association of parent and linkee entries is made on-line.‡

Once a relation has been defined, the user can search the files under that relation. To do this, he uses the command *SEARCH RELATION. The system asks him for the name of the relation, the conditions of the search and the reference value against which the file entries are to be compared. Using the Personnel file and Department file again as the related files and "assignment" as the relation, the user could ask, for example, for a listing of all assignments, by department, of individuals whose last name comes after the reference name "Baker" alphabetically. The appropriate entries will be listed in order.

In addition to the commands for defining a relation and searching on a relation, the system also includes the following commands:

---

† Section V-B describes a system extension which would, in certain cases, permit a user to relate files without naming every associated entry.

‡ Figure 5 illustrates the definition of a relation between two independently existing files, a file containing the name of cities and their characteristics (City file) and a second file containing the names of individuals and information about their home addresses (Home Address file). The relation called "Location," in this case, could be a substitute for the location data field in the Home Address file.

```
*DEFINE RELATIØN                                          -28-6011

LIST:
NAME ØF PARENT FILE
NAME ØF LINKEE FILE
NAME ØF RELATIØN
ØRDER  FIELD

FØR EMERGENCY EXIT DURING INPUT ØF ABØVE PRESS 'BREAK' ØNCE.

CITY
HØME ADDRESS
LØCATIØN
NAME

DØ YØU WANT INSTRUCTIØNS*     YES

                        AUTØMATIC MØDE

A CØDE NUMBER WILL BE PRØVIDED FØR EACH ENTRY IN THE LINKEE FILE. AFTER THIS, THE
ENTRIES IN THE PARENT FILE WILL BE PRINTED ØNE AT A TIME ALØNG WITH SØME FIXED DATA
FIELDS. FØR EACH ØF THESE ENTRIES LIST THE NUMBERS ØF ITS RELATED SUBFILE ENTRIES.
THE NUMBERS MUST BE RIGHT JUSTIFIED WITHIN THE INDICATED FIELDS. TØ TERMINATE INPUT
FØR A GIVEN ENTRY LEAVE A FIELD BLANK AND PRESS 'CARRIAGE RETURN.'

                        MANUAL MØDE

LIST THE NAMES ØF THE PARENT ENTRIES FØLLØWED BY THE NAMES ØF THEIR RELATED SUBFILE
ENTRIES. TØ TERMINATE THE LIST ØF SUBFILE ENTRIES LEAVE A LINE BLANK. TØ TERMINATE
INPUT LEAVE ANØTHER LINE BLANK. WAIT FØR THE WØRD 'READY' BEFØRE TYPING IN EACH
GRØUP ØF PARENT AND LINKEES.

WHICH MØDE DØ YØU WANT* MANUAL

READY

WØBURN
ALLEN MARGAR
ATHANS MICHAEL
CØRR DAVID F

READY

CAMBRIDGE
ANDERSØN ALL
CØHEN MITCHE
CURTISS ARTHUR
FALB PETER L
```

Fig. 5. Example of response to *DEFINE RELATION.

```
READY                                                    -28-6011

CØNCØRD
ANDREWS MARI
ARMENTI AMEDIØ
BLATT HØWARD
CRAIG JEAN G
DICKSØN STUART
FAIETA RITA

READY

BEDFØRD
ARNØLD CHARLES
BARCK PETER
BØYCE SHIRLEY
CRØWTHER THØMAS
DØDGE DØUBLAS

DØDGE DØUBLA IS NØT A LINKEE ENTRY. TYPE IN CØRRECT NAME.

READY

DØDGE DØUGLAS

DØDGE DØUGLA IS NØT A PARENT ENTRY. TYPE IN '*G00F' AND CØNTINUE FRØM THE PØINT
ØF ERRØR.
*G00F
ARLINGTØN
ARNØLD NANCY
BAYNES WILLIAM
CIANCIØLØ LAWRENCE
DAVIS RØBERT
FELDMAN JERØME

READY
```

Fig. 5. Continued.

(1) *LIST RELATIONS. The system lists all relations that are defined in the system.

(2) *DESCRIBE RELATIONS. The system prints out a description of the relation specified by the user.

(3) *FIND PARENT. The system asks the user for the linkee file name, the name of an entry in that file, and the name of a relation defined on the file. It then returns with the name of the parent entry under the given relation. For example, if the file name is "Personnel" the entry name is "Doe, John," and the relation is "assignment," it would return with a parent name like "Department 28."

(4) *FIND LINKEE. This command is the dual of *FIND PARENT. The system asks for the corresponding items of information and returns with the names of all the linkee entries of the given parent. For example, if the file name is "Department," the entry number is "28," and the relation "assignment," the system will list all the individuals assigned to Department 28.

(5) *RELATE ENTRY. This command provides the user with a means of associating a new entry (or set of entries) with parents under an existing (already defined) relation. The system asks for the name of the relation, the name of the parent entry and the names of each entry to be associated with that parent.

(6) *DELETE RELATION. The system asks the user for the name of the relation to be deleted. The files which are related continue to exist in the system but the given relation is removed.

D. Commands For Manipulating Data Fields

These commands permit the user to add a new data field to a file, delete a data field, or update a field value.

To add a new data field to an active file, the user types the command *DEFINE DATA FIELD. The user is asked to give the name of the file, the name of the new data field and the data field type (i.e., integer, floating point, etc.). If the system cannot find the file name in the list of active files, or if the data field name already appears for the file named, the system prints an error comment and asks the user to try again. Otherwise, the system assigns space in each entry of the file to the new data field and adds the name to the list of data field names for that file.

To delete a designated data field from an active file, the user types the command *DELETE DATA FIELD. The user is asked to type the name of the file and the name of the data field to be deleted. The name field of a file can only be deleted by deleting the entire file, and a data field which serves as the ordering field for a relation can only be deleted after the relation has been deleted. The system will notify the user if he attempts to violate these uses. Except for these cases, the system responds to the command by returning data field space to available storage and removing all references to the data field name in the basic files.

To replace a designated data field value with a new value from the console, the user types the command *DEFINE FIELD VALUE. The user is asked to provide the name of the file and the data field name. The system prints the name of each entry and the specified field, and the user then types in the new data values.

E. Block Data Transfer Commands

For some purposes (e.g., for file changes or for small personal files), it is sufficient for the user to update entries or add entries directly from the console. However, in most cases,

the inputs to be stored into a file come in large quantities. This would typically be true, for example, of data acquired in laboratory field experiments where the volume of data recorded would be very high. In such cases, the user wants to have the data read directly into his file from special tape recordings. The *READ CARDS command is used for this purpose. For this command, the (present) system views each tape record as being a card image, i.e., containing no more than 80 columns of punched card information.

To enter a new file into the system the user proceeds in two steps:

(1) The file is defined using the *DEFINE FILE command. At this point, the system has full information on the file and its constituent data fields. The file exists as an empty file (no entries).

(2) The user gives the command *READ CARDS. The system then asks the user to provide the number of the data tape (card file),† the name of the (defined) file into which the data is to be read, and the locations, in terms of card columns, of the data fields on the external card image.

Figure 6 is an example of an exchange between the user and the system in response to the command *READ CARDS. In this example, data are to be read into a file called "Visitors." The Visitors file was defined as file with 11 data fields. These were called: Name, Author. No., Organ. Code, Badge No., SPB 'X', CL, Purpose, Date Granted, Expires, Citizenship, and VI. Code. The data tape may be viewed as a prestored deck of cards with the values appearing in specified card locations. Upon execution of the * READ CARD COMMAND, the user identifies the data fields applying to his data tape. This information is given to the system in the form of a control card prepared on-line at the console.

The system types the starting column number of a data field (beginning with column 1) and the user types the terminal number for the field. He then identifies the field by name. To advance along the card to a particular card column, the user types the number immediately preceding the desired column and strikes the carriage return key in place of the field identification. To return to a previous card position, the user types any number which is less than the current column number. These provisions permit the user to move along the card to select the required field locations or to move backward on the card to correct an erroneous label. If the card column assignment exceeds the data field length originally defined for the file, the system ignores the excessive columns. The data fields, as they apply to the data tape, need not be in the same order as the data fields originally assigned to the file. All that is important is that the data fields of the data tape be properly identified. The system also checks any relation for which the data file is a linkee. For each record of the card file, the user can identify the parent entries for each relation in the same way he identifies data fields. The names of each relation appear at the head of the card file tape. Suppose, for example, the Visitor file of Fig. 6 was related to a file called "Company" with Company as the parent file. Let us call the relation "employer." Then "employer" would appear at the head of the card file tape containing data to be read into the Visitor file. The record for each visitor on the card file would have the name of his company in place of one of the card fields. The system would then treat the company name as the name of a parent entry in the company file and would automatically relate each visitor entry to the appropriate company entry under the relation "employer."

The command * WRITE TAPE causes all of the user's file set to be written onto a specified tape.

---

† In the CTSS system, this number actually references a "pseudotape" maintained on the auxiliary disk storage.

```
*READ CARDS                                                    -28-6012
PLEASE GIVE NAME ØF CARD FILE        VISIT BCD
PLEASE GIVE NAME ØF SYSTEM FILE      VISITØR
I HAVE TRAVERSED THE CURRENT FILES WITHØUT FINDING YØUR FILE.
DØ YØU WISH TØ NAME AGAIN      YES
PLEASE GIVE NAME ØF SYSTEM FILE      VISITØRS
THE NAME 'NAME' IS RESERVED FØR THE PRINCIPAL DATA FIELD
USE CAR. RET. FØR NAME ØF AN UNUSED FIELD


   CARD CØLUMNS 01  TØ  32
   CØNTAINS DATA CALLED NAME

CØLUMNS 13 TØ 32 WILL BE IGNØRED
   CARD CØLUMNS 33  TØ  37
   CØNTAINS DATA CALLED AUTHØR. NØ.
   CARD CØLUMNS 38  TØ  48
   CØNTAINS DATA CALLED ØRGAN. CØDE

CØLUMNS 44 TØ 48 WILL BE IGNØRED
   CARD CØLUMNS 49  TØ  40
   CARD CØLUMNS 38  TØ  48
   CØNTAINS DATA CALLED
   CARD CØLUMNS 49  TØ  50
   CØNTAINS DATA CALLED ØRGAN. CØDE
   CARD CØLUMNS 51  TØ  55
   CØNTAINS DATA CALLED BADGE NØ.
   CARD CØLUMNS 56  TØ  56
   CØNTAINS DATA CALLED SPB 'X'
   CARD CØLUMNS 57  TØ  58
   CØNTAINS DATA CALLED LEVEL ØF CLEARANCE
LEVEL ØF CLE DØES NØT APPEAR IN CURRENT FILES.  IS IT MISSPELT   YES
   CARD CØLUMNS 57  TØ  58
   CØNTAINS DATA CALLED CL
   CARD CØLUMNS 59  TØ  66
   CØNTAINS DATA CALLED PURPØSE

CØLUMNS 65 TØ 66 WILL BE IGNØRED
   CARD CØLUMNS 67  TØ  72
   CØNTAINS DATA CALLED DATE GRANTED
   CARD CØLUMNS 73  TØ  78
   CØNTAINS DATA CALLED EXPIRES
   CARD CØLUMNS 79  TØ  79
   CØNTAINS DATA CALLED CITIZENSHIP
   CARD CØLUMNS 80  TØ  80
   CØNTAINS DATA CALLED VI. CØDE
```

Fig. 6. Example of response to command *READ CARDS.

```
ECHØING                                                    -28-6012
CØLUMNS     NAME          TYPE
 1 -  6 NAME              B C D
 7 - 12 NAME              B C D
13 - 32                   UNUSED
33 - 37 AUTHØR. NØ.       B C D
38 - 48                   UNUSED
49 - 50 ØRGAN. CØDE       B C D
51 - 55 BADGE NØ.         B C D
56 - 56 SPB 'X'           B C D
57 - 58 CL                B C D
59 - 64 PURPØSE           B C D
65 - 66                   UNUSED
67 - 72 DATE GRANTED      B C D
73 - 78 EXPIRES           B C D
79 - 79 CITIZENSHIP       B C D
80 - 80 VI. CØDE          B C D
IS THE ABØVE WHAT YØU WANTED' YES


CØMMAND EXECUTED.


GIVE CØMMAND ØR TYPE *CHØICES
```

Fig. 6. Continued.

III. FILE STRUCTURE AND RETRIEVAL LOGIC

The file structure and retrieval logic of the model was designed to have the following critical features without severely sacrificing time or storage efficiency:

(a) Independence of programs from file structure,
(b) Format variability,
(c) Cross referencing of files (i.e., association between files).

The mechanisms for providing these design features will be discussed in detail in what follows. The following section will treat the questions of time and storage efficiency.

A. Program – File Independence

The independence of programs from file structure was achieved by setting up basic files containing the descriptive information about all files in the system. While this is, in general, a standard technique in system design, the way in which it is implemented in this system is unique. Ordinarily, tables of descriptive information are fixed-format tables made up to hold whatever information the designer might think is necessary for present and future uses of the system.† It is expected that such tables will require little, if any, future changes, and it is therefore assumed to be safe to have the table characteristics appear as constants to the programs which use them. However, except for very specialized systems, this expectation is never realized. Invariably, new requirements get placed on the system and the old tables no longer entirely apply. So both the tables and the programs have to be changed.

Thus, it was necessary to minimize as much as possible the amount of programming changes that would be required by future changes in the tables of descriptive information. We accomplished this by making the tables of descriptive information part of the over-all filing system. The tables are themselves files just like the files a user might create. The descriptive information is treated like ordinary filed data in the system.

The files containing the descriptive information are called the "basic" files and will be described in detail later. These basic files not only contain descriptions of other files in the system, they also contain descriptions of themselves. Because they are stored as data, the file descriptions can be retrieved, modified, or updated just as any other data. Except for a very small set of essential information within the basic files, changes can be made to the descriptive information without altering the data storage and retrieval programs. The file structure and programs are, therefore, virtually independent of any reorganization a user or system designer might contemplate.

B. Entries, Cells, and Linking

The next two design features, format variability and cross referencing of files, are brought about primarily by the use of list structures. The basic filing unit is an entry. Physically, an entry can be made as large as necessary by linking together standard-sized portions of storage, called "cells." Similarly, a file can be extended to any desired length by linking new entries to those already in the file. The relations between files are implemented through list structures that tie entries in one file to entries in another. Through the use of relations, the user can effectively organize his files to suit his convenience.

---

† The "TABLE SHOP" in LUCID and the "Dictionary" in Ref. 3.

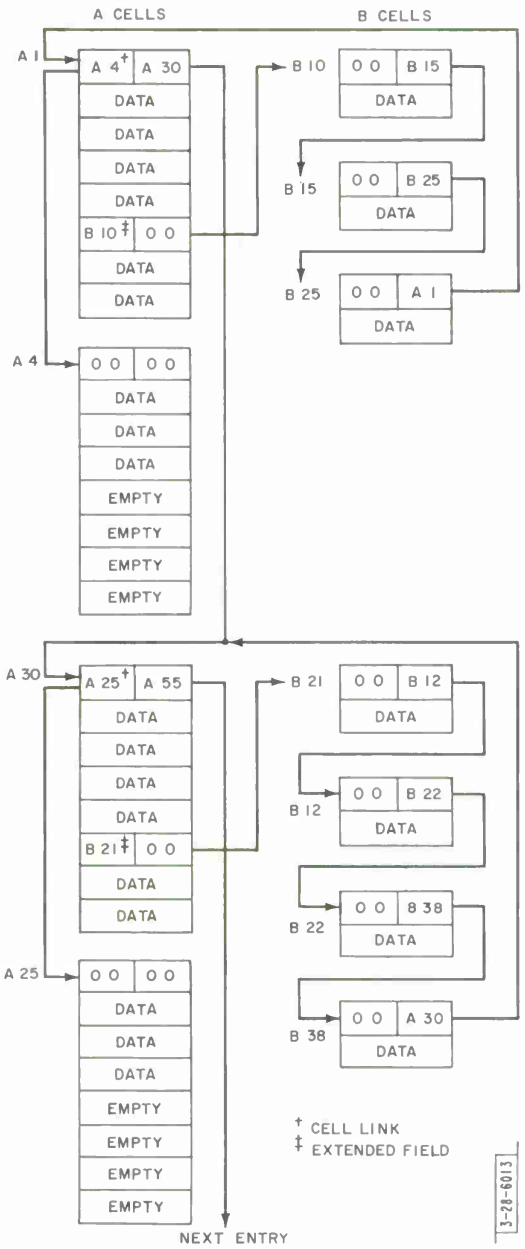


Fig. 7. Sample file.

These features will be made clear in the system description that follows.

All files are constructed by putting together standard portions of storage units of fixed length, called cells. Each cell has a number which uniquely identifies it. The system presently uses two kinds of cells, one eight registers long (A cells) and one two registers long (B cells). While the number of cell types and cell lengths are constant in this version, there is nothing inherent in either the file structure or retrieval logic which precludes making them parameters. In constructing a file, the system selects the minimum number of cells necessary.

When a user defines a new file, he is asked to provide the system with the name and coding of each field. From this information, the system determines the number of cells needed for an entry in the file. The number of cells per entry is stored as an item of descriptive information about the file. When data is put into the file, the system gets the cell needed from available storage† and creates the entries. If more than one cell must be used for an entry, the system stores a link which ties the cells. Variable length fields are handled in the same way. A link in the field position addresses the first member of a string of cells. Each member of the string is similarly tied by links. In this way, a field in an entry can be extended to fit any content length. Figure 7 illustrates a simplified version of a file. The file has two cells per entry and one field is extended.

C. Basic Files

Within the system, the format of an entry is determined entirely by the names of the fields, their positions in the entry, the coding required by each and the position of links which either extend a field, tie one cell of an entry to another, or tie one entry to another entry. This format information is contained in one of three basic files, (see Fig. 8) the Active file, the Data Field file, and the Relations file, which the system maintains. These will be described in some detail below. It is important to recall here, however, that the user need only know the name and coding of each field in his file to set it up.‡

The first of the three basic files in the system is called the "Active" file. The Active file contains an entry for every file in the system including the basic files themselves. Each entry has, as its data fields, the name of a file, the cell number of the starting entry of the file, the number of cells each entry of the file uses (a single number), the file type, and a list of unused positions in an entry (called "empty space"). The file type specifies one of two types in the current system: a "named" file or a "numbered" file. This simply distinguishes a file whose entries are named alphabetically from one whose entries are numbered. The distinction is useful for searching on files, but is not necessary. The empty space field is an extended field.

The Data Field file contains an entry for each data field of the files listed in the Active file. An entry in the Data Field file has as its data fields, the name of a data field, the position of the data field in the entry, and the field coding. Since the Data Field file is itself one of the Active files, the Data Field file contains entries for its own data fields. The field coding identifies the coding of the data values that will appear in the data field. The present version of the system allows six types of coding: BCD (alphanumeric), integer, and floating point, and lists of

---

† All empty (unused) cells are maintained by the system as push-down lists, one for each size of cell used (two in the present system).

‡ In a fully developed system, the user could supply other items of information, such as the range within which a data field value may fall. But we see no need to require format information except for checking and special output purposes.

A CELLS

ACTIVE FILE

	1		2		3	4	5	6	7		8			
	CELL†		ENTRY†		NAME†	FILE-START†	CELLS/ENTRY	TYPE-OF-FILE	FIELDS/FILE		PARENT REL'S		LINKEE REL'S	EMPTY SPACE
1	4	1	2	2	(Dummy Entry)									
2	4	2	2	3	ACTIVE-FILES	1	1	0	1	7	1	18	7 -	7 -
3	4	3	2	4	DATA--FIELDS	5	1	0	1	10	7	- 0	18	1 0
4	4	4	4	2	RELATIONS	18	2	0	1	5	7	- 1	19	1 4

DATA FIELD FILE

	CELL†		ENTRY†		NAME†	FIELD PLACE	FIELD TYPE	UNUSED	UNUSED	FIELDS/FILE			
5	4	5	2	6	ASCEND-PLACE	24	1			2	6	4	4
6	4	6	2	7	BRANCH-PLACE	14	1			2	8	4	4
7	4	7	2	8	CELLS/ENTRY	10	1			2	9	4	2
8	4	8	2	9	DSCEND-PLACE	16	1			2	13	4	4
9	4	9	2	10	EMPTY-SPACE	16	4			2	12	4	2
10	4	10	2	11	FIELD-PLACE	8	1			2	11	1	3
11	4	11	2	12	FIELD-TYPE	10	1			2	10	4	3
12	4	12	2	13	FILE-START	8	1			2	17	4	2
13	4	13	2	14	ORDER-FIELD	12	0			2	14	4	4
14	4	14	2	15	ORDER-RULE	10	1			2	15	4	4
15	4	15	2	16	RELAT'N-TYPE	8	1			2	16	4	4
16	4	16	2	17	RETURN-PLACE	26	1			2	5	4	4
17	4	17	4	3	TYPE-OF-F FILE	12	1			2	7	4	2

RELATIONS FILE

	CELL†		ENTRY†		NAME†	RELAT'N-TYPE	ORDER-RULE	ORDER-FIELD	BRANCH-PLACE	DSCEND-PLACE
18	2	21	2	19	FIELDS/FILE	2	0	0	13	15
19	2	22	2	20	LINKEE-REL'S	3	0	0	15	30
20	2	23	4	4	PARENT-REL'S	3	0	0	14	27

	CELL†			UNUSED		ASCEND-PLACE	RETURN-PLACE	PARENT-REL'S						LINKEE-REL'S			
21	4	18				-	16	2	19	3	20	4	2	2	18	3	18	4	3
22	4	19				31	32	2	20	3	18	4	2	2	20	3	20	4	4
23	4	20				28	29	2	18	3	19	4	2	2	19	3	19	4	4

†FIXED FIELD POSITIONS

3-28-6055

8 CELLS

	CELL		ENTRY		DATA
0	4	0	2	1	11
1	4	1	2	2	12
2	4	2	2	3	13
3	4	3	4	3	14
4	4	4	2	5	18
5	4	5	2	6	19
6	4	6	2	7	20
7	4	7	2	8	21
8	4	8	4	4	22

LINK KEYS

0 POINTER
1 BRANCH
2 DESCEND
3 ASCEND
4 RETURN
5 UNUSED
6 (SPARE)
7 EMPTY

FIELD-TYPE

0 BCD
1 INTEGER
2 FLTG. POINT
3 BCD LIST
4 INTEGER LIST
5 FLTG. POINT

RELATION TYPE

0 ONE WAY LIST
1 TWO WAY LIST
2 ONE WAY RING
3 TWO WAY RING
4 ONE WAY GROUP

ORDERING RULE

0 ALPHABETIC
1 NUMERICAL
2 ARBITRARY
3 SPECIAL

TYPE-OF-FILE

0 NAMED
1 NUMBERED

Fig. 8. Basic files.

these three types. The latter three types of data fields are treated as extended fields which branch to a list.

D. List Structures

Before describing the Relations file, something must be said about the list structures employed in the system. Since the early use of list structures (e.g., in Ref. 4) as a means for converting a fixed-address memory into a variable-address storage space, there have been a number of variations of list structures proposed, fitted to special organizations of data. Perlis' "threaded list" structures[5] and Weizenbaums' "knotted list"[6] are examples of such variations.

Restricting the system to any single list structure could place serious constraints on possible file formats. These considerations led us to adopt a number of simple conventions which would permit us to construct any list structure we might need.

The linking of objects (such as entries) in our system reduces, essentially, to five kinds of links which must be distinguished to permit the system to traverse the files most effectively. A link which associates an entry in one file with the subset of entries which make up its subfiles, we call a "branch." (In general, this linking is equivalent, in list structure terminology, to the linking of a header with a sublist.) A link which ties the subset of entries to its parent is called a "return." A link which ties entries within a file (or list) in one direction is called a "descend" link and in the opposite direction an "ascend" link. Finally, a branch link to a single member subfile is called a "pointer." These five link types provide all that is required to tie components of files in ways which make them readily accessible to the system routines. A 3-bit key identifies the link type. The key also is used to identify an unused link space and an empty list. Figure 9 summarizes the link nomenclature for the system.

Using these five kinds of links, we have constructed list structure types. It was intended that the system monitor itself. Depending on the file processing history, the system would incorporate the list structure which would optimize the storage and retrieval process. A simple list structure involving only a one-way tying of file entries would be adequate for processing relatively short files, but inadequate for lengthy files. For example, consider a search by the system programs of a very long subfile where the first entry has been found to meet all the search conditions. It would be wasteful to then have the system programs traverse the entire

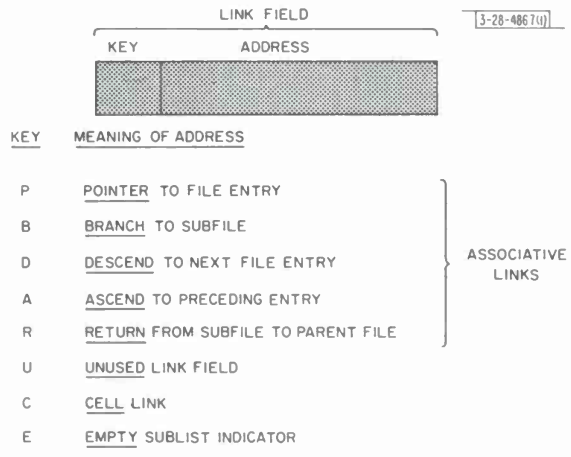


Fig. 9. Link code.

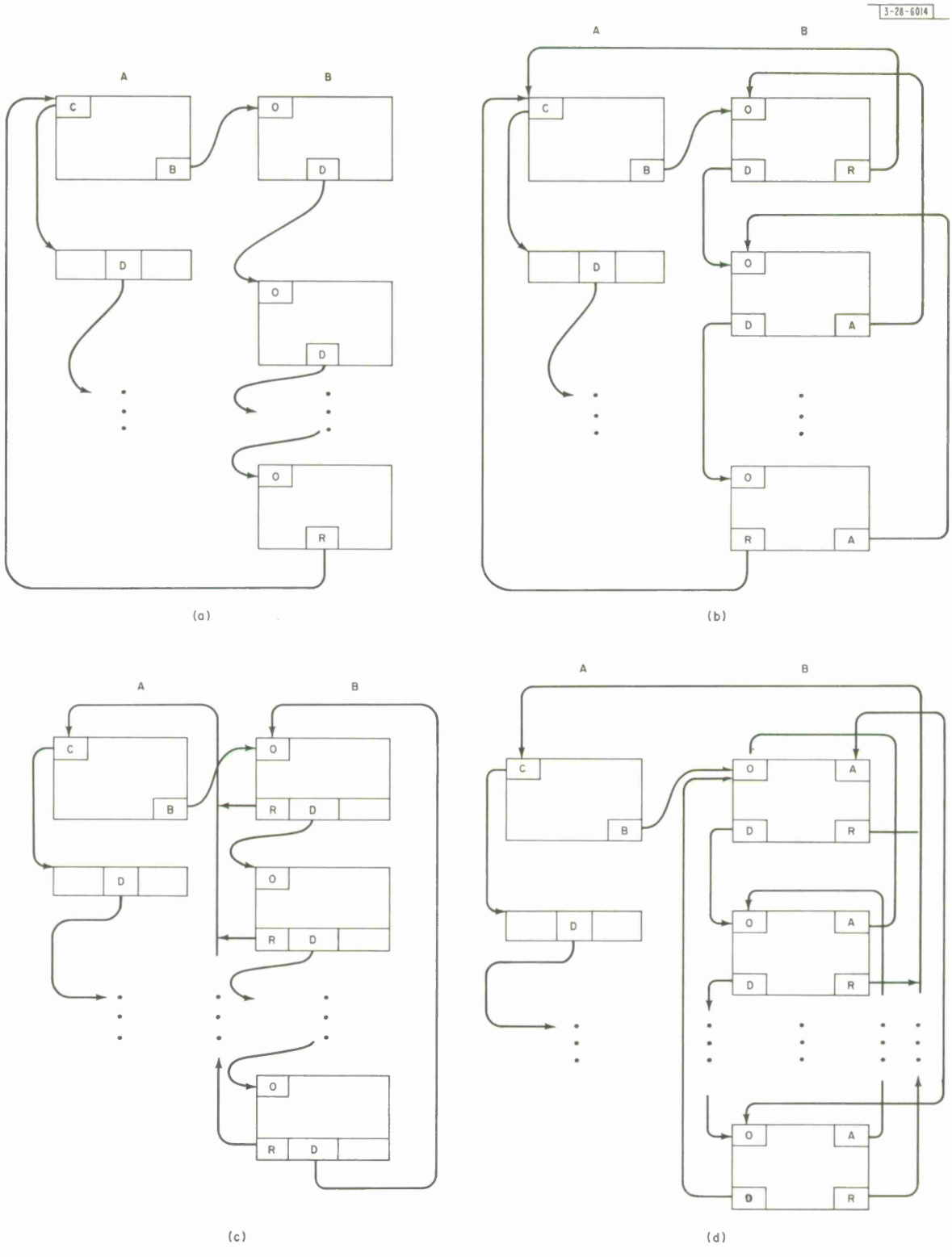


Fig. 10. List structures: (a) One-way list; (b) Two-way list; (c) One-way ring; (d) Two-way ring.

subfile to find the return link to the parent entry. At some point, it clearly becomes worthwhile to change from a simple list structure to a structure having return links at strategic places in the file. By maintaining a record of the operations performed on the various files, the system could periodically decide, on the basis of this record, whether to convert from a simple list structure to one of more complexity, or, on the contrary, to convert from a complex structure to a simpler one.

Monitoring programs do not exist in the present model. For purposes of illustrating their uses, however, the different list structures have been incorporated into the basic files.

Figure 10 illustrates the four types of list structures used in the system.

A string of entries linked by either a descend link or ascend link is a one-way list. If the entries are linked by both a descend and ascend link, the string is called "a two-way" list. The string of entries consisting of a parent entry in a file A with a pointer or branch to a list of entries in a file B is called a "simple" list. In a simple list, the last entry of subfile B may have an "end of list" code (EOL) or a return in the descend link position.

A list whose members are linked successively to each other is a "ring." Any member of a ring can be designated the "first" member. A simple list whose subfile entries form a ring with returns from each entry to the parent is called a "threaded" ring. If the links which form the ring are descend links only, the list structure is a "one-way threaded" ring. If the list structure uses both ascend and descend links, it is a "two-way threaded" ring. In the present model, we have adopted the convention that all rings will be threaded rings. We shall therefore refer to all such list structures as one-way or two-way rings.

E. Implementation of Relations

A relation defined by a user is implemented internally by the system as one of the four types of list structures. All relations initially appear in the form of a simple list. The list structure type is assigned a code, called the "relation type," which is entered as one of the items of descriptive information in the Relation file, the third of the basic files. There exists an entry in the Relations file for each defined relation in the system, including those defined on the basic files themselves. In addition to the relation type, the Relations file carries the name of every relation, the ordering rule for that relation, the name of the data field on which the ordering is done, and the relative positions within entries of the links used to implement the relation.

The cross-association within the basic files themselves can be seen in Fig. 8, already referenced. In Fig. 8, the basic files contain only information describing themselves. Each line is a cell. The A cells are numbered from 1 to 23, the B cells from 0 to 8.† If a new file were defined, the system would use the free A cells beginning with the next number available from the list of available cells to add an entry in the Active file and entries in the Data Field file for each data field. If a new relation were defined, the system would get two available A cells and add an entry to the Relations file. Each of the three basic files is organized as a simple list with the entries linked on the entry link and ordered on the name field. By convention, every file has this minimum association. For this reason, no relation name has been given to this association. The associations between the Active file, the Data Field file, and the Relations file, however, have been given relation names and entered in the Relations file. The subfile relation between the Active file and the Data Field file is called "Fields/File" and is structured as a one-way ring.

† In practice, of course, the cell numbers need not be consecutive, since they are joined as a link structure.

There are two relations associating the Active file and the Relation file. One of these is called "Parent Relations" (Parent-Rel's) and it associates entries in the Active file with entries in the Relations file for which they are parents. The other is called "Linkee Relation" (Linkee Rel's) and it associates entries in the Active file with entries in the Relations file for which they are linkees.

F. Modification of Basic Files

While all of the information presented in the basic files is regarded as essential to the operation of the system, only a small part need be fixed in position. The files marked with a dagger (†) in Fig. 8 are the data fields whose positions must be fixed for the system. Otherwise, the basic files may be altered like any other file in the system. For example, we presently foresee the possibility of including the acceptable range of values for a data field as one of the pieces of information describing a data field. This would be implemented as a new data field in the Data Field file. The user (system designer) would implement this change on-line by using the standard commands to define a new data field for an existing file. To make this change, the system will first determine if there is enough empty space in a Data Field file entry, as currently organized, to permit the inclusion of the new data field. If the key in the empty space link indicates that a word position is available, the system will assign the data field to that word position. The new data field information will be added as an entry in the Data Field file in the same way as with any new entry for a file. If no word positions are available, the system will enlarge the Data Field file by adding a second cell to each entry. The first position in the new cell will be given the name of the new data field.

Notice that the above steps for augmenting the basic files are performed by the same procedures used in altering any file in the system. The basic files contain sufficient information descriptive of themselves to allow their own extension. Figure 11 illustrates the use of information from the basic files by the system routine for defining a new data field to be added to a file. In Fig. 11, the numbers listed next to the logical steps in the flow diagram indicate the information used by the routine as indicated in the accompanying list.

As one can see from this example, the procedures for processing the files, no matter how complex, rest on a relatively small number of operations. The principal routines in the system simply search the basic files for descriptive information. The various processing routines, therefore, like the routines for searching files, searching on relations, defining a new file, or deleting a file, are basically independent of the file formats or file contents. Searching for the descriptive information is carried out in the same fashion as searching for any other data. The starting location of the Active file is fixed. Within an Active file entry, the position of the data field giving the starting location of every file (including the important case of the Data Fields file) is fixed. Within the Data Fields file, the position of the field giving the field position for any data field in the system is also fixed. With this information, the system programs can search any one of the basic files and get any information needed to proceed further. Thus, we have effectively built a "boot-strapping" feature into the system to allow for future system modification.

Furthermore, we have chosen a file structure, i.e., an organization based on data fields, entries, and file relations, which is broad enough to accommodate any form of content arrangement. These features make file additions, deletions, and general file reorganization possible

without the need to modify system programs. No matter what changes are made to the alterable characteristics of the files, we are assured that no routine in the system will be affected.

IV. SYSTEM EFFICIENCY

One should distinguish between system utility and system efficiency. The utility of the system to the user is based mainly on the versatility of the system in providing data storage and retrieval functions to match his specific (and changing) information requirements. The flexibility of the system in matching the user's needs has been illustrated in the preceding sections. Such versatility of operations can only be achieved at some cost in processing time and storage requirements over that which would be required in the (hypothetical) situation where the form of data to be stored and the class of queries to be serviced were completely predictable and unchanging. It is of interest to quantitatively relate the present system's efficiency with this extreme case so as to verify that time and space requirements are not prohibitive and to illustrate the dependence of time and space factors on the volume and other characteristics of the data to be stored.

A. Storage Efficiency

Storage efficiency is mildly dependent on the choice of cell size. Too large a cell size results in extra waste of storage due to partially filled cells. Too small a cell size results in extra waste of storage due to the need for more cell linking. It can be shown that the optimum single cell size is the square root of the average volume of information (data and links) to be stored per entry. Efficiency can be further improved, of course, by using two or more cell sizes as standard. In the present system, we have used eight-word cells for all file entries and two-word cells for extended fields.

The efficiency of the system in use of storage can be considered in two parts: first, the storage required for the basic files and second, the overhead storage required in the data files themselves for cell linking, associative linking (for the relations), and unused space (due to partially filled cells).

The number of words required for storage of the basic files is given by

$$202 + 8F + 8D + 16R$$

where

F = number of user's files
D = number of user's data fields
R = number of user's relations.

The 202 words represent the initial form of the basic files† (before any user data are introduced) and hence describe only themselves. These very moderate requirements are well justified by the data file-program independence they provide. Note that the basic file storage requirements are independent of the number of entries per file, due to the general fixed-format rule used.

The overhead storage required in the data files themselves can be computed on an entry basis. For a fixed cell size the fractional overhead in cell linking (and entry linking within a file) is constant. Assuming $\frac{1}{2}$ word per link, one finds that the fractional overhead is 6.25 and 25 percent respectively, for the 8- and 2-word cells. Since most of the data are stored in 8-word

† See Fig. 8.

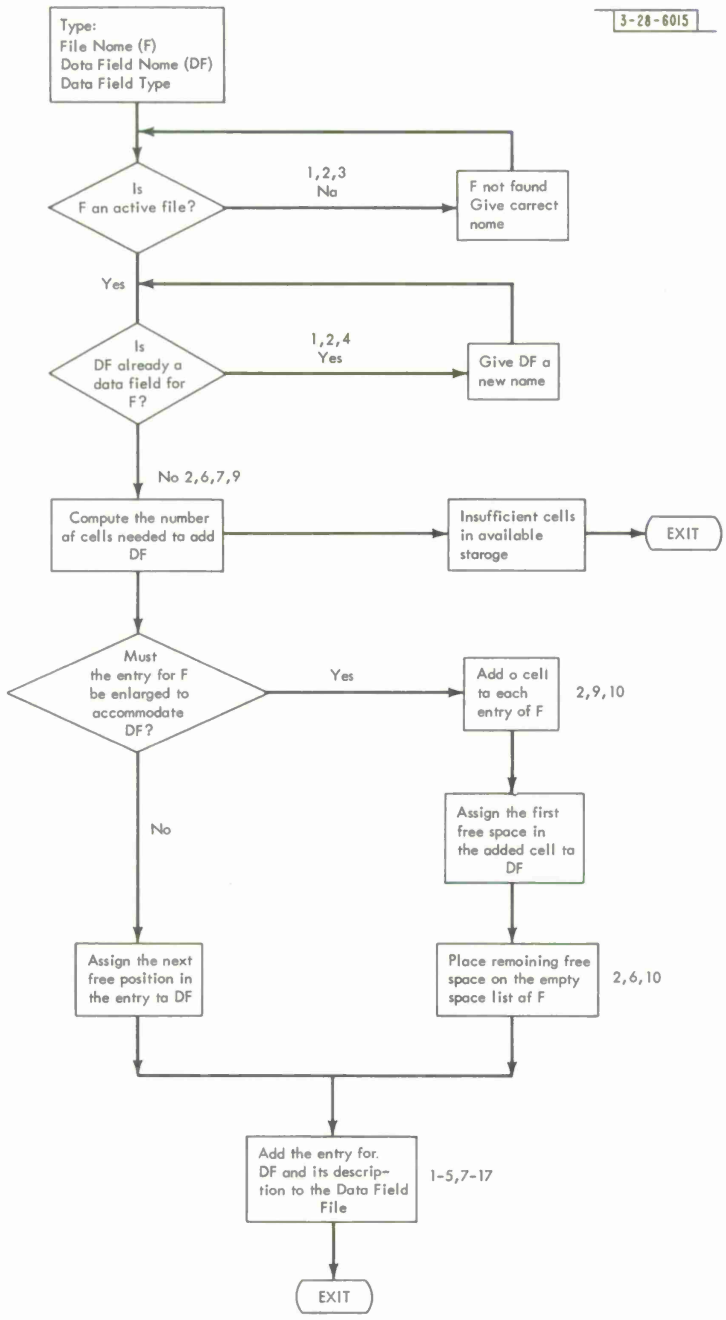


Fig. 11. Define Data Field routine.

Descriptive Information Used by Routine

(1) Word position of name field in files. (Same for all files.)†

(2) Word position of entry link. (Same for all files.)†

(3) Location of entry F in Active file.

(4) Starting location of Data Field file.

(5) Word position of field type in Data Field file.

(6) Word position of empty space in Active file.

(7) Word position of cells/entry in Active file.

(8) Location of entry for Data Field file in Active file.

(9) Starting location of file F.

(10) Word position of cell link. (Same for all files.)†

(11) Starting location of Relations file.

(12) Word position of relation type in Relations file.

(13) Word position of ordering rule in Relations file.

(14) Word position of ordering field in Relations file.

(15) Word position of branch place, descend place, ascend place, and return place (link positions) in Relation file.

(16) Word position of field place in Data Field file.†

(17) Word position of file start in Active file.†

† These items of information are in a fixed position. All others are variable in location and are located by using the former.

Fig. 11. Continued.

cells, the general overhead for cell linking is at most 10 percent. This overhead is clearly justified since, by allowing relocation of all storage cells, the dynamic storage allocation problem is essentially eliminated.

The percent of storage for associative linking, required to implement relations between files, depends on the ratio k of the number of relations defined on a file to the number of data fields in the file. The percentage is then 50-percent k. Thus, for one relation for each four data fields (k = 0.25), an additional $12\frac{1}{2}$ percent in storage is required. Such a ratio would represent a high degree of cross-association in large files.

Finally, the relative amount of storage unused due to partially filled cells decreases geometrically with the number of data words stored per entry in a file set (D/F) and is equal to (400F/D)%. Thus, for a file set in which the average number of data fields per file is 40, the overhead in storage due to unfilled cells is 10 percent.

Thus, for most file sets, one can expect a total storage overhead in the range of 10 to 50 percent of the volume of data to be stored, varying according to the number of relations defined and the size of the file entries.

B. Time Efficiency

The capability for the user to cross-associate his data files in arbitrary ways by use of relations provides not only convenience in structuring files to suit the user's personal interests but also the means for rapid file operations. As has been illustrated, the use of relations allows (among other associations) construction of file-subfile hierarchies to any number of levels. These multi-level associations are equivalent in form and utility to what, in list processing terminology, are called "tree structures," with the exception that in conventional list processing each point (node) in the tree structure would typically contain one symbol, whereas in our system the node corresponds to a complete entry of data. This distinction provides the additional advantage that nodes (entries) may belong to many different tree structures and the branching criteria may depend upon the values of any of the data fields in the entry.

For searching and modifying large files, tree structures combine the best features of tabular and list organizations of data. In tabular organizations, entries are stored in a known ordered form in sequential physical blocks of storage. Given a reference value to be matched with the ordering field, the search can be conducted rapidly by repeated subdivisions of the entry space. Search times for tabular organizations of data therefore require on the average $\log_2 N$ steps to locate the desired entry, where N is the total number of entries. However, to alter the files, e.g., by adding or deleting entries, half the file contents (on the average) must be moved, to retain the ordered sequential storage; this time cost grows linearly with N. In any large system using tabular organizations and requiring both file access and dynamic modification, the data movement times constitute the bulk of the time requirement.

Conventional list organization of data, on the other hand, avoids data movement operations for file changes but requires searching to be conducted in a serial fashion. In list organizations, the physical location of any entry bears no relation to the ordered position of the entry within the file set. Entries are chained together in order by stored link addresses; addition or deletion of an entry (once it is located) requires a negligible amount of processing time which is independent of the size of the file set. However, since searches must be conducted serially, search times grow directly with the number of entries N (compared with $\log_2 N$ for tabular organizations) and for large files the time delays in searching would be prohibitive.

In tree structure organizations of data, entries are also located at arbitrary physical locations and linked by stored addresses to eliminate time requirements for altering file contents. Rather than a simple list structure, however, the entries are organized into a many-level tree which can be traversed more rapidly in search operations. The tree structure can be characterized by a parameter b, the branching factor, which is the average number of nodes on the next lower tree level associated with any node of the tree. In our system, in which the tree is actually a multi-level file-subfile hierarchy, the branching factor corresponds to the average number of entries associated as a subfile with any entry by the defined relations which form the tree. With data structured in this way, search times vary according to $(b/2) \log_b N$, where b is the branching factor and N is again the total number of entries in the file set. Thus, the capability to organize files in tree structures provides both ease in altering of the files and search times close to the minimum possible.

V. SYSTEM EXTENSIONS

The currently operating model of the system has satisfactorily demonstrated the versatility of the filing and retrieval techniques and the ease of on-line user control of the system operations. Many extensions of the command language are possible at this point. This section describes some of these extensions and the steps required in their implementation. It should be noted that none of the extensions described below require any modification of the basic files or of the present set of storage and retrieval programs.

A. Group Relations

In the experimental model, a relation between files requires that the subsets of entries in the linkee file be disjoint. A linkee entry cannot have more than one parent under any one relation This is clearly a restriction we want to remove since in many instances a user will want to associate parent entries in one file with overlapping subsets of entries in a linkee file. In creating a bibliography, for example, one may want to link different reports with authors, where each author may have multiple reports and each report may have multiple authorship. We call a relation of this type, which associates overlapping subsets of entries in a linkee file with entries in a parent file, a "group" relation.

One way of implementing such a relation within the framework of our model is illustrated in Figs. 12 and 13. The system would create a dummy file consisting of pointers which would tie together the overlapping subsets of file C to file A. The special "pointer" file B would not appear as a named file in the Active file. The system would recognize its existence by the relation code number. Figure 13 shows how the dual of the file in Fig. 12 would be created. The dual would tie the entries of C back to A by way of entries in the same pointer file B. One would very likely want the system to create the dual automatically with every group relation so as to provide an easy two-way searching path.

B. Automatic Association

In the current model, when a user defines a new relation on two active files, he must specify to the system the name of each entry and its associated subfile entries. Similarly, for the case where entries are being added to a file which is related to other files, the user must specify, by name, the parent or linkee entries in those files. This entry-by-entry association can be

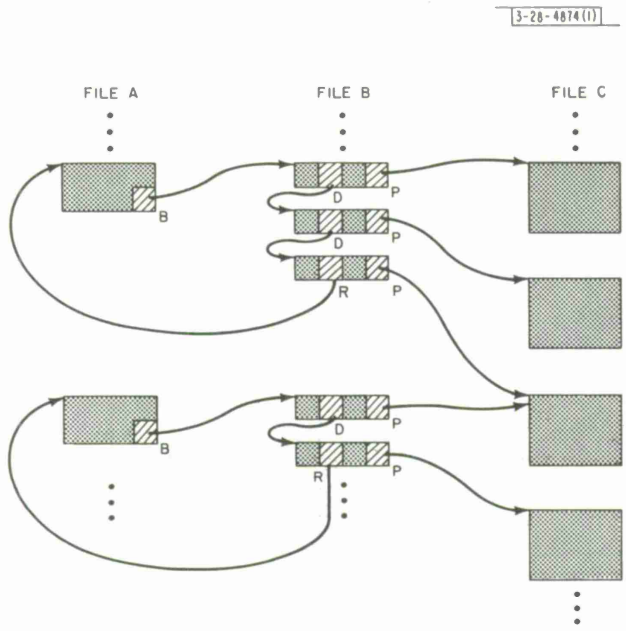
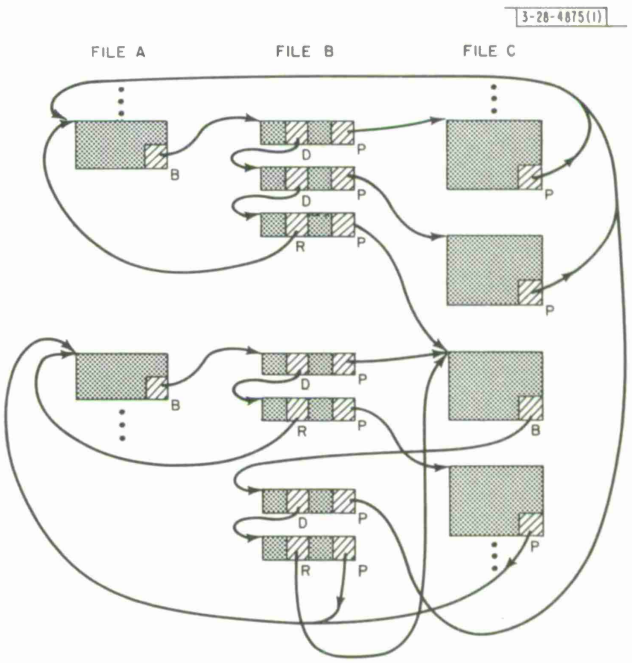


Fig. 12. Group relation.

Fig. 13. Group relation with linking.



eliminated for a number of special cases. In these cases, the user can be permitted to relate files in terms of conditions placed on the two files rather than by designating individual entries. This can be done whenever an unambiguous mapping can be made between entries in the two files. A relation can always replace data fields which are duplicated in two files. If, for example, department assignment is a data field in a personnel file and a file of departments also exists, then the data field in the personnel file can be deleted and a relation defined which associates each man with his respective department. The determination of which personnel entry must be associated with which department file entry is, of course, trivial since it simply involves a matching of the values in the assignment field with the department names.

Again if the relation to be defined maps a parent entry onto ordered subsets of the linkee file, then the requirements for setting up the relation can be stated in the form of simple conditions. For these cases, routines would be added to those which respond to the relation definition command. In addition to asking the user the names of the files and the relation name, the system would also ask the condition on which the relation is to be made.

Consider, for example, two files, one which contains the slots in a table of organization for a military unit and one which contains the personnel for that unit. A user who wants to relate the personnel to the table could define the relation in terms of a condition on the military rank of each man in the file. Thus, he could specify for each slot the rank or range of the personnel who fill it.

C. Tree Searching

The system, at present, provides the means for tree structuring (see Sec. IV-B) of the data files by repeated use of relations. To search such trees, however, the user must, for each of the levels in the tree, specify the relation and search conditions involved. A desirable extension is that of providing special commands for constructing and searching such trees.

By use of the present command set, the user can index subject matter and form hierarchical structures of the data files. The indexes (intermediate levels in the trees) may be files which already exist or may be added to the system for that purpose with the *DEFINE FILE command.

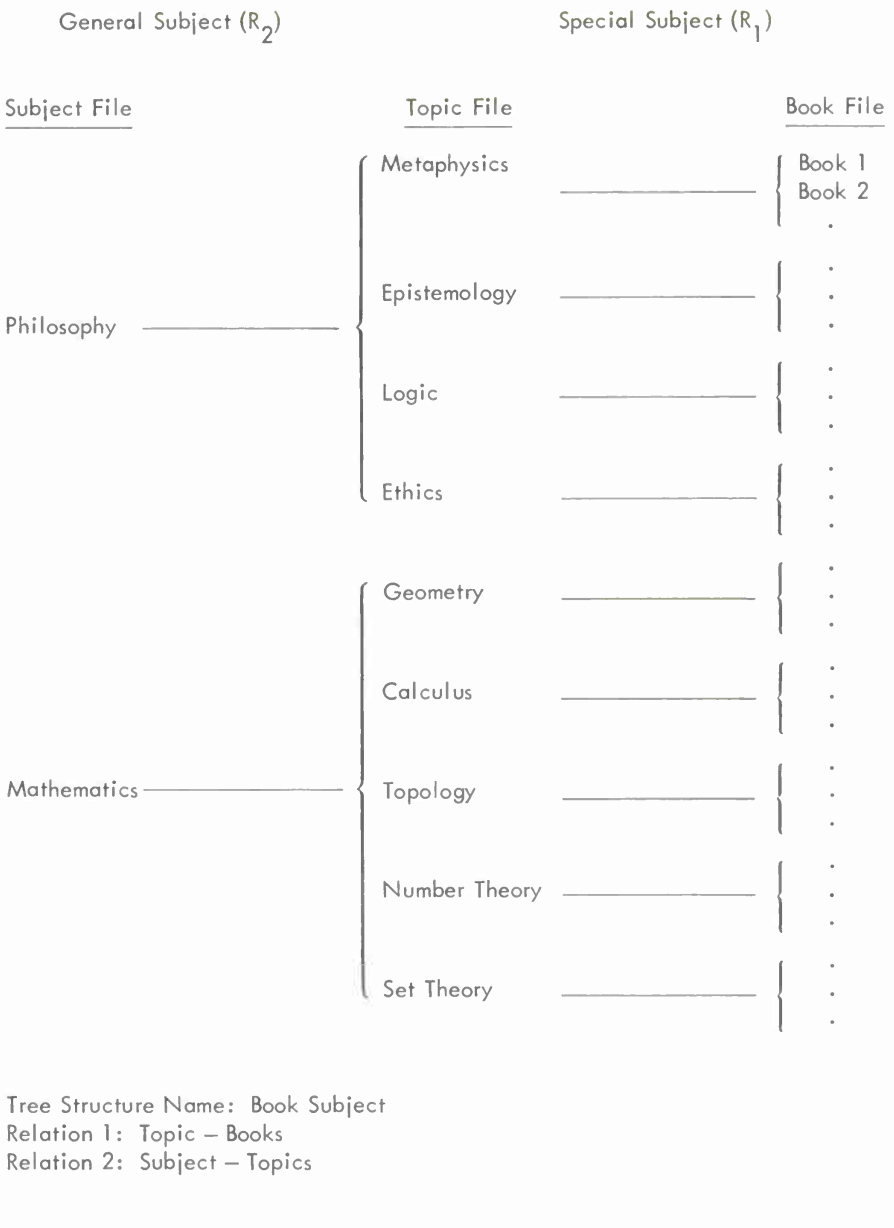
By use of the *DEFINE RELATION command the user can then define a relation R_1, which associates the file in question with the index file. The user would next partition the index file in a similar fashion, creating a new set of category names and a new relation R_2. This procedure could be repeated to any desired level for relations R_3, $R_4, \ldots, R_n$. The system would then have, in its relations file, n relations relating n + 1 files in a tree or hierarchical structure.

To illustrate this procedure, consider a file of library catalog cards and a tree structure which divides the set of books named by the cards into topics and divides the topics into more general subject categories. For simplicity, we assume no overlap in the categories, i.e., the category names unambiguously partition the entries in the card file. (With the group relation feature such overlapping would be possible.) Figure 14 shows a part of the tree structure.

Having constructed such tree structures, the user can locate selected subsets of the entries by repeated application of the *SEARCH RELATION command. Of course, when the present command is set, the appropriate file names and relation must be specified by the user.

A natural extension, allowing more facility at tree manipulation, is to identify and record an entire tree by name. This can be done for example by defining a relation "TREE" which associates the file upon which the tree is based with the set of n relations which form the (n + 1)-level tree.

3-28-6056



Tree Structure Name: Book Subject
Relation 1: Topic – Books
Relation 2: Subject – Topics

Fig. 14. Catalog card file.

This tree structure could now be given a name and entered into a file containing the tree name, each of the relations defining the tree, and any other information the system might need to search such a tree. To search through the tree, the system would go to the tree structure file, find the appropriate tree, and systematically make a search by way of each of the relations which defines the tree.

In this scheme, the index would identify the tree by name as one of the means for locating subject matter specified in the query. Thus, in responding to a query asking for the author, publisher, etc., of books on topology, the system would find the tree named "Book-Subject," and then go to the tree file to determine how it should search that tree structure. It would first search on the relation called "Subject-Topics" until it found topology and then search on the relation "Topic-Books" to get the requested information on each book.

New commands and associated programs can then be added to the system which would provide the following additional services for the user:

(1) Automatic association of the file and relation set with the tree designation given by the user.

(2) On-line description of the tree structure including the intermediate-level categories.

(3) Execution of search requests including any number of restricting conditions on categories and subcategories.

(4) Statistical summarization of counts of entries under selected portions of the tree structure.

D. Macro Commands

An additional feature of great utility to the user is the capability to ask the system to remember extended sequences of commands and to execute them on demand. Among other uses, this feature would provide automatic report generation on the present contents of selected portions of the files.

This feature can be implemented by new commands which allow the user to enter and leave a "record" mode during which the system would retain a record of the command sequence. Upon leaving the record mode, the user would assign a new command name to the sequence which could then be used at any later time to execute the entire command sequence. Facilities can also be provided to allow the user to review, edit, and modify previously defined macro commands.

NOTE

The research described in this report was accomplished by the following staff personnel: A. W. Armenti, J. A. Arnow, D. E. Hall, D. A. Koniver, J. F. Nolan, H. C. Peterson, and P. M. Wortman.

REFERENCES

1. F. J. Corbato, "System Requirements for Multiple Access, Time Shared Computers," MAC-TR-3, Computation Center, M. I. T. (1962).
2. F. J. Corbato, et al., "The Compatible Time Sharing System, A Programmer's Guide," Computation Center, M. I. T. (1963).
3. J. A. Terrasi, "7090 File Generation System," W-4861, MITRE Corporation (18 April 1962).
4. A. Newall and F. Tonge, "An Introduction to Information Processing Language V," Commun. ACM 3, 205 (1960).
5. A. J. Perlis and C. Thorton, "Symbol Manipulation by Threaded Lists," Commun. ACM 3, 195 (1960).
6. J. Weizenbaum, "Knotted List Structures," Commun. ACM 5, 161 (1962).

ADDITIONAL REFERENCES

E. Franks, "LUCID System of Automatic Programming Directly from Data Processing System Design Specifications," FN 6797/000/00, Systems Development Corporation (9 August 1962).

_____, "LUCID Control System Design," TM 1749/101/00, Systems Development Corporation (27 January 1964), Vol. I.

J. J. Croke and W. K. Rawdon, "Query Languages in Data Retrieval Systems: Final Report," TM-3727, MITRE Corporation (15 September 1963).

V. E. Giulano, et al., "Automatic Message Retrieval," Arthur D. Little, Inc. (November 1963).

A. Oettinger and S. Kuno, et al., "Mathematic Linguistics and Automatic Translation," Report No. NSF-8, Computation Laboratory, Harvard University (January 1963).

A. K. Wolf, C. S. Chomsky, and B. F. Green, Jr. , "The Baseball Program: An Automatic Question-Answerer," Technical Report 306, Lincoln Laboratory, M. I. T. (11 August 1963), Vols. I and II, DDC 432038.

DOCUMENT CONTROL DATA – R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

Lincoln Laboratory, M. I. T.

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP
None

3. REPORT TITLE

An Experimental On-Line Data Storage and Retrieval System

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Technical Report

5. AUTHOR(S) *(Last name, first name, initial)*

Nolan, John F. Armenti, Amedio W.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
3 February 1965	44	12

8a. CONTRACT OR GRANT NO. AF 19(628)-500
Office of Naval Research Contract Nonr-4102(01)

b. PROJECT NO.

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
Technical Report 377

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
ESD-TDR-65-36

10. AVAILABILITY/LIMITATION NOTICES

None

11. SUPPLEMENTARY NOTES

None

12. SPONSORING MILITARY ACTIVITY
Air Force Systems Command, USAF

Advanced Research Projects Agency, Department of Defense

13. ABSTRACT

This report describes an experimental program system designed to test and demonstrate on-line storage and retrieval of formatted data based on complete internal descriptions of the files. The use of internal description allows each user (who need not be a trained programmer) to define, modify, and cross-associate data files to suit his particular needs. The experimental program system was implemented by remote use of the Compatible Time-Sharing System facilities of Project MAC at the Massachusetts Institute of Technology.

14. KEY WORDS

data storage
information retrieval
time sharing
computer
data processing
rings
tree structures

Printed by
United States Air Force
L. G. Hanscom Field
Bedford, Massachusetts